Behavioral and Dynamic Security Functions Chaining For Android Devices

Gaëtan Hurel, Rémi Badonnel, Abdelkader Lahmadi, Olivier Festor

Behavioral and Dynamic Security Functions Chaining For Android Devices

Gaëtan Hurel, Rémi Badonnel, Abdelkader Lahmadi and Olivier Festor

INRIA Nancy Grand-Est - LORIA, France
Email: {gaetan.hurel, remi.badonnel, abdelkader.lahmadi, olivier.festor}@inria.fr

***Abstract*—We present an approach for dynamically outsourcing and composing security functions for mobile devices, according to the network behavior of their running applications. Applications are characterized from a network point of view using data mining and clustering techniques with the aim to select their appropriate security functions. Software-defined networking mechanisms are employed to chain the selected functions and to redirect mobile apps traffic through the resulting security compositions. Those ones can be fully outsourced or split between *in-cloud* and *on-device*. Both a prototype and extensive simulations demonstrate the feasibility of the approach and assess its benefits.**

I. Introduction

Mobile devices such as smartphones and tablets are now widely spread and used in our everyday life. Android is currently the leading operating system in the world for mobile devices with up to 75% of market share [4]. Its official application store (Google Play) offers more than 1.4 million apps as of February 2015, and this number shows an exponential increase since 2009 due to the open nature of the Android ecosystem [5]. Regular users generally trust apps from the official market and are not fully aware of the privacy and security threats that they may bring [23][24]. In the meantime, it is difficult even for experts to predict the behavior of a running app [19], even though the set of required permissions for this app is known [30]. From a network point of view, this observation is emphasized due to the low granularity of the Android permission framework: a specific permission (INTERNET), when granted, allows any app to use the Internet connectivity regardless the servers to be reached or the data to be transmitted. In the recent years, a significant number of on-device and cloud-based approaches have been proposed to reduce the threats induced by buggy or malicious apps on mobile devices [15]. However, very few of them focus on fundamental network security. Furthermore, most of these solutions do not adapt the security intelligence (e.g. policies) and the associated processing to the communication patterns of the running apps.

In order to fill this gap, we propose in this paper a new approach to compose security functions for mobile devices according to the type and the network behavior of their running apps. To this end, security functions are chained together by leveraging the Software-Defined Networking (SDN) paradigm and the Openflow protocol. The resulting compositions can be fully outsourced in the network, or use an hybrid in-cloud/on-device deployment scheme. In order to know which security functions should be used for which app(s), we extract the network behavior of these latter by employing data mining techniques on a dataset of network flow information collected using a dedicated monitoring platform [22]. We show that our proposition is feasible and permits to keep the device battery usage at a low level, while still delivering dynamic and appropriate network security. Our main contributions are: i) a strategy for clustering mobile apps depending on their network behavior and for choosing security functions accordingly, ii) a mathematical model for defining and partitioning security compositions between in-cloud and on-device, and iii) a running prototype and extensive experiments to evaluate our approach.

The rest of this paper is organized as follows: Section II gives an overview of our approach and its challenges. Section III details our data mining and clustering strategy to characterize the network behavior of mobile apps. Section IV describes the mathematical model behind the generation of the security compositions. Prototyping and evaluation are tackled in Section V. We present related work in Section VI, and give conclusions and perspectives in Section VII.

II. Problem Statement

Our approach is illustrated by Fig. 1 and has already been presented in [20]. It involves three actors: (i) the mobile device to protect which runs several apps and a lightweight agent, (ii) a cloud provider infrastructure hosting security functions, and (iii) the remote destinations that the apps are communicating with. When a running app on the device wants to communicate over the network, the corresponding traffic may be redirected - in both direction - by the device's agent through a composition of security functions in order to be protected.

Challenges and research questions

Our first objective is to define and build appropriate security compositions for mobile devices depending on their context and risks. Since in this work compositions focus on network communications, we believe such context and risks may be strongly tied to the network behavior of the mobile apps which are running on the devices. A first step consists therefore in getting a better understanding of how mobile apps behave from a network point of view in order to get an idea of the security checks to apply. As a second objective, we aim at optimizing the deployment of the compositions between the mobile devices and the cloud infrastructure according to different factors. The rationale behind outsourcing mobile security functions is mainly to save resources and battery usage on the devices. In the meantime, blocking communications at the device level before being transmitted through the network may induce a substantial gain in resources saving on the device (e.g. battery and CPU) due to the avoided transmission cost for instance [12]. Such blocking may occur under network attacks or unwanted communications, when an app running on the

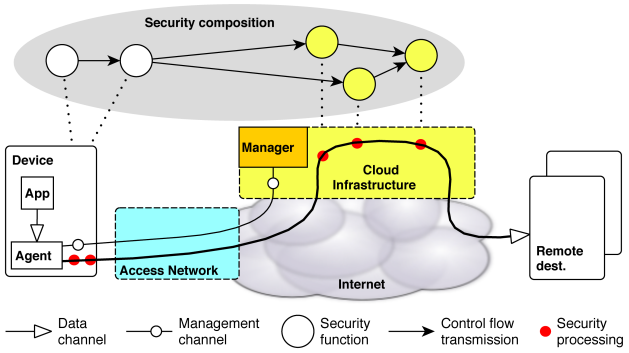


Fig. 1: Proposed approach for delivering cloud-based mobile security chains.

device does not behave as expected. Therefore, partitioning the security composition consists in deploying the associated security functions between the device and the cloud infrastructure, in such a way that resources savings on the device are optimized for a given level of security. For the remaining of this paper, we call a *cut* the action of partitioning a composition into two sub-compositions. It is worth noting that a cut (and the associated partitions) can change over time according to some dynamic parameters (e.g. device battery level).

The contribution of this paper aims at answering the following questions: *What security functions to select in order to fit the security requirements for a given mobile app? How to cut the resulting composition in order to optimize the deployment of its functions between the device and the cloud?* Answers to each of them are given respectively in Sections III and IV.

III. Mobile apps network communication patterns

Our idea is to adapt the security functions and resulting compositions to the apps which are running on the device according to their network behaviors. We describe in this section our strategy based on data mining and clustering techniques to reach this goal.

A. Apps network flows collection and aggregation

We have previously developed a monitoring platform capable to collect, store, analyze and visualize log and network data related to running Android apps [22]. This platform relies on a set of on-device probes to monitor network and system activities of the apps. We focus in this work on the network flows only, which are collected using the NetFlow protocol [2]. We aggregate this data to obtain additional information (e.g. unique count of server addresses) and to prepare features for clustering the apps according to their network behavior.

B. Selected network features

According to [31] and [14], network traffic generated by Android apps can be classified into i) origin traffic - i.e. the traffic that comes from/to the servers owned by the app providers - ii) CDN/Cloud traffic - i.e. the traffic that comes from/to CDN or cloud providers such as Akamai or AmazonWS - iii) Google traffic - i.e. the traffic exchanged with Google servers such as *.1e100.net - and iv) third-party traffic - i.e. the traffic that comes from/to advertisements and analytics networks. The authors also propose the use of additional metrics to profile apps such as the number of traffic sources, the ratio of traffic sent/received, and the split of HTTP/HTTPS traffic. In our work, we make the choice of taking all of these metrics as features for the clustering analysis. We add however the following changes: i) we decide to split third-party traffic into advertisements traffic and analytics traffic, ii) we add the number of different server ports fetched by an app, and iii) since some apps like Facebook Lite don't use traditional ports (i.e. 8000 instead of 80/443), we consider the global percentage of HTTP, HTTPS and other ports traffic rather than the split of HTTP/HTTPS traffic. In the end, the selected features for clustering apps are the following:

- traffic in/out ratio
- unique count of server addresses
- unique count of server ports
- % of origin traffic
- % of CDN/cloud traffic
- % of Google Service Framework (GSF) traffic
- % of advertisements traffic
- % of analytics traffic
- % of HTTP traffic
- % of HTTPS traffic
- % of other traffic

It is worth noting that for determining the nature of the traffic (i.e. origin, CDN/Cloud, etc.), we employ manual techniques such as hostname resolution and whois queries.

C. Mobile apps clustering

We give here a description of our first scheme for clustering apps according to their network behavior. Since all of our defined features are numeric only and the dispersion of the data is significant, we have decided to compare several clustering algorithms which are a) hierachical clustering (average-linkage), b) K-medoids and c) Self-Organizing Maps (SOM). For these three algorithms, we use the Euclidean distance as the distance function and we normalize all features using min-max normalization. Though this methodology is mostly practical and needs a validation step, yet it yielded to satisfying first results in terms of apps clustering. While hierarchical clustering is used to estimate an optimal number of clusters, K-medoids and particularly SOM ease their visualization. The dataset we work on was collected from 5 personal mobile devices during one week. At the end, we have extracted 6 clusters for a total of 45 mobile applications and around 800000 network flows. Extracted clusters using the K-medoids algorithm and their main characteristics are listed in Table I.

D. Mapping app clusters to security functions

As reflected by the last column of the Table I, we consider the use of the previously found clusters and their network characterization in order to select appropriate security functions for building compositions. If we take as example the cluster 6 that we labeled "Gaming & multimedia", one may want to use a firewall (FW) for blocking advertisement traffic, one or more Intrusion Detection/Prevention System (IDS/IPS) to check the unencrypted traffic for potential network attacks, as well as a data leakage prevention engine (DLP) for controling the information sent to the analytics servers. One possible resulting composition is discussed further in section V and is acting as

TABLE I: Extracted clusters using K-medoids and examples of associated security functions.

Cluster	Label	Main characteristics	Security functions
1	Google services framework	Origin traffic, mostly HTTPS.	FW (e.g. whitelist)
2	Google & social networks	Mostly origin traffic, some CDN; fully HTTPS.	TLS proxy, IDS/IPS
3	Mobile device management	Half of Google traffic, half of Orig/CDN; no HTTP(S) at all.	FW, DPI
4	Messaging (e.g. whatsapp)	Mostly CDN, balanced traffic ratio; mix of HTTP(S) and other ports.	FW, IDS
5	Utilities (e.g. meteo)	Mainly CDN and analytics traffic; mostly HTTP.	FW, DLP
6	Gaming & multimedia	Mainly CDN, but also advert. and analytics traffic; mostly HTTP.	FW, IPS and DLP

referential for our evaluation. Other examples of compositions can include security functions such as Deep Packet Inspection (DPI) and TLS proxy, e.g. for encrypted traffic like in [28]. Currently, the mapping between clusters and security functions is done in a manual way. Also, the sequence (order) of the selected functions within a composition is an important point that we plan to investigate further for future work. In the next Section, we address the problem of partitioning such compositions between the device and the cloud.

IV. Partitioning mobile security compositions

In this section, we first present in details the modeling of the security compositions. We then describe how we use the resulting model in order to efficiently partition the compositions between the device and the cloud.

A. Composition graph

A security composition C can be formalized by a directed acyclic graph that we call composition graph. A composition graph $G_C = (V, E)$ is made of:

- a set $V = \{sf_1, sf_2, ..., sf_n\}$ of vertices, where each vertex represents a security function of the composition;
- a set $E = \{tc_1, tc_2, ..., tc_m\}$ of control edges, where each edge expresses the control flow transmission between its corresponding security functions; it is worth noting that such transmission can be sequential, conditional or concurrent.

1) Properties on vertices: Each vertex sf_i is assigned one or more weights expressing several costs for processing *n* flows related to the app(s) for which the composition is built. For the remaining of this paper, we use the term *network communication* for denoting these *n* flows. The costs are:

- $W_{cpu}(sf_i)$, the average cost w.r.t. CPU usage;
- $W_{batt}(sf_i)$, the average cost w.r.t. battery usage;
- $W_{delay}(sf_i)$, the average cost w.r.t. processing time (i.e. treatment delay);

As we make the assumption that cloud resources are unlimited, those weights are only suitable for security functions which are run on the device. Additionally, we introduce two metrics denoting the potential rewards of running a security function locally in case this one blocks a communication before being transmitted (α and β respectively express the CPU and battery cost required for a communication transmission):

- $\sigma_{cpu}(sf_i) = \frac{\alpha^t}{W_{cpu}(sf_i)}$ w.r.t. CPU usage savings;
- $\sigma_{batt}(sf_i) = \frac{\beta^t}{W_{batt}(sf_i)}$ w.r.t. battery usage savings

It is worth noting that treatment delay for a blocked communication does not make sense in our context. Thus, we assume for the remaining of this paper $\sigma_{delay}(sf_i) = 0 \ \forall i$.

2) Properties on edges: Each control edge is assigned one weight, W_{info}, expressing the control communication overhead due to the amount of security information to be transmitted to the following security function. This security information typically includes the previous function(s) results in order to give the next one all the necessary information to adapt its processing logic on the network traffic to analyze. The network traffic itself is not included in this cost, since it is always sent only once to the outsourced infrastructure.

B. Composition partitioning

We describe here our strategy for efficiently partitioning security compositions between *on-device* and *in-cloud* deployment. For the remaining of this section, we call C_{dev} (respectively C_{cld}) the partition of the initial composition C which will be run on the mobile device (respectively in the cloud infrastructure).

1) Composition, partitions and cut-set: Given a security composition C and its associated composition graph $G_C = (V, E)$, a partition P can be represented as a sub-graph $G_{C_{part}} = (V_{part}, E_{part})$ of G_C with $F_{part} \subset F$ and $T_{part} \subset T$. The cut-set T_{CS} induced by the partitioning of C into $P1$ and $P2$ gathers all the edges which are linking the two partitions; that is, $T_{CS} = T - (T_{P1} \cup T_{P2})$.

Two properties must be enforced for a partition graph $G_{C_{part}} = (V_{part}, E_{part})$ in order to be potentially chosen as C_{dev} or C_{cld}. First, $G_{C_{part}}$ must be a connected sub-graph of G_C. Second, in order to avoid round-trip between the device and the cloud infrastructure, the edges of the associated cut-set T_{CS} must all go in the same direction: that is, from $G_{C_{part}}$ towards the other partition, or vice versa.

2) Partition costs: Several costs can be assigned to each potential partition of G_C according to the different weights and rewards of the included vertices. We define these costs as W^P_{cpu}, W^P_{batt} and W^P_{delay}. According to the control transmission schemes that occur among the vertices, the way to calculate these costs differs. The rules 1 to 3 below summarize our method for computing each of those costs for a partition $G_{C_{part}} = (V_{part}, E_{part})$ such that $F_{part} = \{sf_a, ..., sf_k\}$. δ represents the rate of communications to be blocked and can be estimated once the network behaviors of the apps to protect are known, as introduced in Section III.

$$W^P_X(G_{C_{part}}) = \sum_{i=a}^{k} (W_X(sf_i) - \delta.\sigma_X(sf_i)) \quad (1)$$

$$W^P_X(G_{C_{part}}) = max(W_X(sf_i), ..., W_X(sf_k)) \quad (2)$$

$$W^P_X(G_{C_{part}}) = \sum_{i=a}^{k} (p^V_i.(W_X(sf_i) - \delta.\sigma_X(sf_i))) \quad (3)$$

Rule 1 allows to compute the W^P_{cpu}, W^P_{batt} and W^P_{delay} of $G_{C_{part}}$ when sequential control transmission only is used.

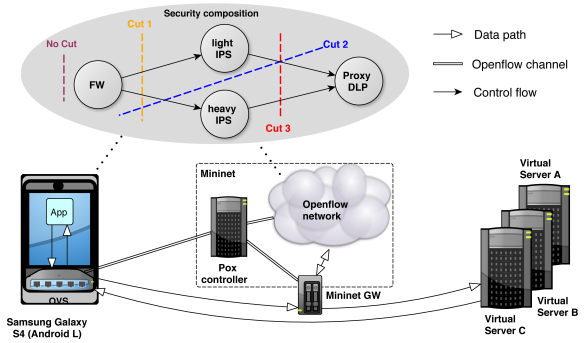


Fig. 2: Testbed and experimental setup

When this latter is concurrent, rule 1 permits to calculate W^P_{cpu} and W^P_{batt} while rule 2 permits to determine W^P_{delay}. When the control transmission is conditional, rule 3 is used to compute all the costs of $G_{C_{part}}$. This last rule consists in pondering the given weights and rewards of each vertex according to its probability p^V of being executed. We use the methodology presented in [17] and based on Markov chains for determining the execution probability of conditional vertices in a composition. When different transmission schemes are used in a same partition, the associated rules can be combined in order to compute the related costs.

An additional cost W^P_{info} is also assigned to the cut-set $T_{CS} = \{tc_a, ..., tc_k\}$, based on the weights W_{info} of the edges contained it contains. This cost represents the overhead of control information to be sent or received by the device (according to the direction of the traffic). Rules 4 and 5 are used to calculate the W^P_{info} associated to a given cut, provided that the edges of the cut-set employ either sequential/concurrent (rule 4) or conditional transmission scheme (rule 5, with p^E_i the probability of the edge t_i of being traversed):

$$W^P_{info}(G_{C_{part}}) = \sum_{i=a}^{k} W_{info}(tc_i) \quad (4)$$

$$W^P_{info}(G_{C_{part}}) = \sum_{i=a}^{k} p^E_i.W_{info}(tc_i) \quad (5)$$

3) Choosing the best partition: Once the costs associated to the potential partitions of a composition are known, the following step is to choose efficiently C_{dev} (and C_{cld} accordingly). This decision step can be tackled using at least two different strategies. The first one is to use a simple approach where each cost of the selected C_{dev} (e.g. W^P_{batt}) must respect an associated constraint (e.g. W^{const}_{batt}), typically specified by an end-user or an administrator. The second option consists in turning the decision step into an optimization problem similar to [13], where the authors use Integer-Linear Programming (ILP) to find the optimal tradeoff between the resources savings dues to the offloading process (i.e. W^P_{batt}, W^P_{cpu}, W^P_{delay}) and the network overhead due to the transmissions of control information between the partitions (i.e. W^P_{info}). We leave for future work a deeper study of different optimization algorithms in order to compare their performances.

V. Prototyping & Evaluation

We detail here the prototyping and the evaluation of our approach with respect to the composition partitioning problem.

A. Prototype and testbed

The prototype we developed relies on:

- a Samsung Galaxy S4 device with a custom CyanogenMod ROM (12 Nightly intl) running Android Lollipop (5.0.1). The device lightweight agent shown in Fig 1 is implemented by an OpenvSwitch (OVS) version 2.3.1. In addition, a dedicated Android application is installed on the smartphone in order to generate significant amounts of HTTP traffic for our experiments.
- a Mininet emulator [8] hosted on a computer with an Intel Xeon 3.70 GHz CPU and 32 Go of RAM, with 16 Go allocated to the Mininet VM. We emulated an OpenFlow-based network containing OVS switches (version 1.10.0) to forward the traffic, and standard Linux hosts to host security functions. We chose POX as the OpenFlow controller, and the traffic redirection and forwarding logic was implemented as a POX module.
- a set of three custom application servers running in the same LAN of the smartphone and the Mininet emulated network for practical reasons. Those three servers are the ones the smartphone is communicating with by the means of the dedicated Android application.

Using this prototype, we built the security composition introduced in Section III and depicted in Fig 2. This composition involves four instances of security functions:

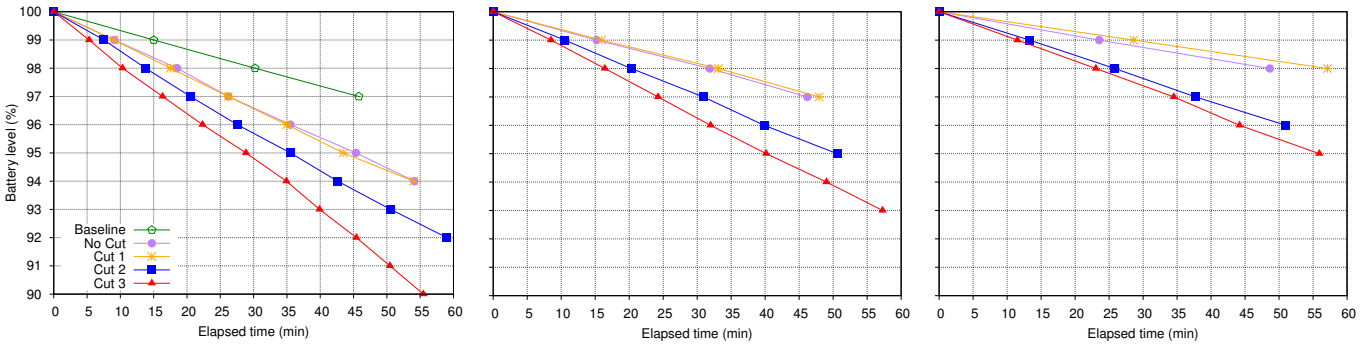
- a firewall implementing IP blacklisting. We chose to use the Linux iptables/netfilter firewall [9] with a public dataset [1] containing around 5000 blacklisted addresses.
- two concurrent Intrusion Prevention Systems (IPS) inspecting network traffic for potential network attack. We chose to use Suricata [11] with the Emerging Threats open ruleset [7]. We splitted that ruleset amongst the two Suricata instances: 4225 rules inspecting network metadata were used on the *light* instance, while 11252 rules inspecting HTTP traffic were used on the *heavy* instance.
- a basic Data Leakage Prevention engine (DLP) inspecting application traffic for potential data leak. We implemented this function using the Squid proxy [10] coupled with the Dansguardian tool [6].

The composition was used for securing the smartphone outgoing traffic only. Redirecting the traffic through the composition was done using IP address rewriting at the device switch level, both for incoming and outgoing traffic. To this end, a specific host within Mininet (Mininet GW on Fig. 2) had an IP address publicly reachable from the smartphone.

B. Experimental setup

We have prealably defined four potential cuts and four associated C_{dev} for the selected composition. Our experiment session focuses on the potential benefits and caveats when using each of these partitions, according to the ratio of communications to be blocked:

- *NoCut* : this configuration consists in employing the composition while outsourcing all of the included security functions in the network; if a blocking rate is specified, the outsourced firewall function is in charge of blocking the required communications.



(a) default, no blocking rate (b) blocking rate = 20% (c) blocking rate = 60%

Fig. 3: Device battery discharging over time for different blocking rates. The y-axis remains the same for all charts.

- *Cut1*: C_{dev} includes the firewall while C_{cld} includes the two IPS and the DLP. The firewall function on the device is in charge of blocking the communications.
- *Cut2*: C_{dev} includes the firewall and the light IPS while C_{cld} includes the heavy IPS and the DLP. The light IPS on the device is in charge of blocking the communications.
- *Cut3*: C_{dev} includes the firewall and the two IPS while C_{cld} includes the the DLP only. The heavy IPS on the device is in charge of blocking the communications.

The four cuts are shown on the Fig. 2. For *Cut 2* and *Cut 3*, we had to cross-compile Suricata using the Android NDK in order to port it on the smartphone. For all of our experiments, the Android application was generating 10 HTTP requests per second among the three remote servers. Though not realistic, we justify this choice by the fact that we needed to generate high load of traffic in order to emphasize the impact regarding resources usage on the device. Additionally, the rate of communications to be blocked was varying among the experiments - from 0% to 100% at intervals of 20%.

C. Experimental results

We now discuss our experimental results regarding impacts on battery, CPU and delay (RTT) from the device side when employing the given security composition.

1) Battery discharging: Our first experiment considers the impact on the device battery induced by the use of the composition, for each potential cut and three different blocking rates. In that context, the dedicated Android app installed on the device generates traffic during one hour for each configuration. We use the Google Battery Historian tool [3] in order to measure the battery level over time. Results for the different configurations and blocking rates are depicted in Fig. 3. The line *Baseline* with empty pentagonal points on Fig. 3a acts as a referential, as it shows the battery discharging during one hour in a basic scenario with neither redirection nor local security functions. We can see a significant difference between this configuration and the ones corresponding to the use of our solution. For instance, while only 3% of battery have been consumed during the first 45 minutes for *Baseline*, 5% and 8% have been consumed in the same time respectively for *NoCut* and *Cut3*. For *NoCut*, we believe this discharging overhead is mainly due to the address rewriting step of the device switch. For *Cut3*, the discharging overhead is accentuated due to the execution of the two Suricata instances on the device. In parallel, we can observe similar behavior for *NoCut* and *Cut1*, from which we can deduce that the firewall footprint is negligible when run on the device. Meanwhile, Fig. 3b and 3c show that the higher is the blocking rate, the fewer is the battery discharging. With a blocking rate starting at 20%, our strategy even can save battery in *NoCut* and *Cut1* compared to the *Baseline* scenario, where no blocking at all is done since no security function is run. This battery saving can be explained by the fact that when a communication is blocked, the IP address rewriting and the transmission cost (e.g. antenna) can be avoided on the device side. For *Cut1*, this gain is emphasized since the communication is blocked before being redirected in the network, in contrast with *NoCut*. For *Cut2* and *Cut3*, this saving is balanced with the discharging due to the execution of the Suricata instance(s) on the device.

2) CPU usage: Our second experiment takes the exactly same setup than the first one but focuses this time on the CPU usage induced by the composition. The measures are taken using the *dumpsys* and *top* tools on the device side while the Android app is generating HTTP traffic. Since netfilter is run at the kernel level, we are not able of measuring its CPU impact. However, the previous results regarding battery discharging make us believe that its footprint is negligible. Results for each configuration and blocking rate are shown in Fig. 4, where the average CPU usage induced by the OVS switch, the light IPS and the heavy IPS on the device are illustrated using stacked histograms. We can observe that the switch footprint is relatively significant (up to 7%), particularly when the blocking rate is null or relatively low. We explain this trend by the fact that when a communication is blocked, IP address rewriting (switch) is avoided and rule matching checks (IPS) is partially done - the IPS don't have to browse their whole ruleset before matching the one which blocks the communication(s). An interesting consequence is when the blocking rate is between 60% and 100% for *Cut1*: as the firewall blocks the communications on the device before being processed by the switch, the impacts of IP address rewriting and data transmission are avoided. The overall CPU usage for *Cut1* being fewer than for *NoCut* - where the data is processed by the switch and sent before being blocked in the cloud - we can conclude that the switch CPU usage is higher than the firewall's one, i.e. $\sigma_{cpu}(FW)$ is relatively high.

3) Application RTT: Our last experiment deals with the application round-time trip (RTT) overhead induced by the composition for each of its potential cuts. The blocking rate for this experiment is fixed as it does not influence the RTT. We measure this latter by adding specific code in the Android

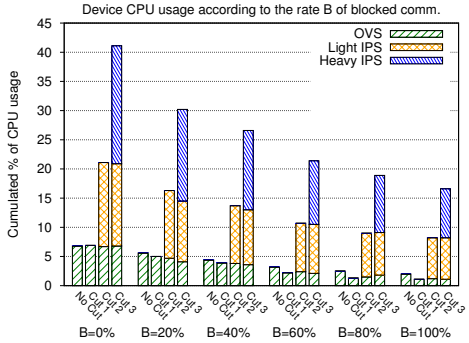


Fig. 4: Average CPU usage per cut and blocking rate.

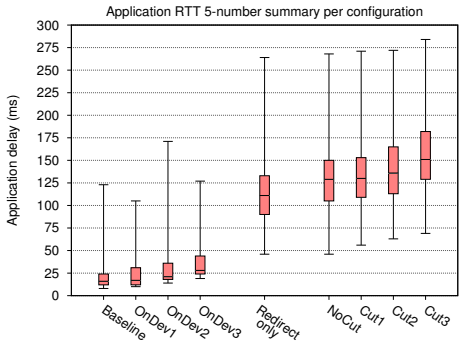


Fig. 5: Application RTT 5-number summary per cut.

application which generates HTTP traffic. Results are shown in Fig. 5 under the form of boxplots for each configuration. Similarly to the first experiment on battery, the *Baseline* boxplot corresponds to the case where our solution is not used at all. *OnDev1*, *OnDev2* and *OnDev3* boxplots represent the case where no composition is used (thus no redirection) but security functions are run on the smartphones (respectively the same ones as for *Cut1*, *Cut2* and *Cut3*). Finally, the *RedirectOnly* boxplot denotes the case where the traffic is being redirected, but no security function is employed. We can observe that security functions, when run on the device, proportionally add a small delay overhead compared to the *Baseline* case. This increase is also observed for the different cuts compared to the *NoCut* case. A key finding is the delay overhead induced by the redirection as shown by *RedirectOnly*. After verification, this overhead proves to be due to the address rewriting step performed by the OVS switch.

VI. Related Work

Mobile security has been subjected to intensive research work these past years [26] [15]. However, most of the work done in this area focus on host-based security. Proposed solutions typically employ three types of architecture, namely i) on-device, ii) cloud-based, and iii) hybrid architectures. Several cloud-based approaches have already been proposed for outsourcing specific security functions [25] [21], though no composition or chaining is done. We already introduced our approach in [20]. Extracting the network fingerprint of mobile apps has already been studied a couple of times [31] [14]. Wei et al. achieve this goal by using an Android-specific version of *tcpdump* on an instrumented smartphone. Dai et al. run the mobile apps within a monitored emulator and use UI fuzzing to build a comprehensive network profile for each application [14]. In our work, we leverage a monitoring platform that relies on on-device probes to gather network information from Android apps [22]. Service outsourcing and chaining is a highly topical issue with the advances in cloud technologies. Sherry et al. propose a solution to dynamically and transparently outsource middleboxes across several cloud providers using virtualization and different redirection mechanisms [29]. Gibb et al. present a similar work where a cloud-based architecture is designed for outsourcing network functionalities [18] using SDN. Regarding the chaining of such network functionalities, Qazi and al. introduces SIMPLE [27], a policy enforcement layer based on SDN and flow correlation for middlebox traffic steering. In the same vein, Fayazbakhsh et al. propose Flowtags [16], an architecture where middleboxes are extended to support Openflow and to deal with dynamic middlebox flow mangling. Perhaps the closest work of ours is [28], where the authors leverage a Network Function Virtualization (NFV) router to provide a per-user policy enforcement on mobile applications through service chaining. Aside from the employed strategy and technologies, a major difference with our work is that they neither tackle the partitioning of the chains nor the network characterization of mobile apps for functions selection.

VII. Conclusions

The plethora of apps available for mobile devices like smartphones and tablets makes their security a critical issue, as most of these systems are strongly connected to the Internet. In this paper, we have proposed an approach based on compositions of security functions in order to inspect communications of mobile devices. In this context, we have defined a set of features to characterize the network behavior of mobile apps and cluster these latter accordingly. We have leveraged the resulting clusters in order to provide the required security functions to compose for checking the apps behavior. Given the resulting composition, we have then addressed its partitioning problem with the aim to find an efficient deployment of the security functions between the device and the cloud infrastructure. To this end, we have put forward a mathematical model for determining the costs induced by a composition on the device side regarding battery and CPU usage, as well as treatment delays and transmission overhead. These costs are used to determine an efficient cut of a composition according to some parameters that may vary over time (e.g. device battery level). We evaluated the potential benefits and drawbacks of our approach through an extensive set of experiments, and showed that additional resources savings can occur on the devices when using our solution under certain circumstances.

For future work, we plan to study different optimization and clustering algorithms in order to compare their performance. In parallel, the way the weights are assigned to nodes and edges in a composition graph remains to be formally tackled. We finally want to consider the case where several partitions have to be deployed on a device running different groups of apps, possibly using graph merging algorithms in order to avoid conflicts or redundancies between the partitions.

Acknowledgments

This work was partly funded by the Flamingo Network of Excellence project (ICT-318488). The authors also thank the AKD STIC-AmSud Project.

REFERENCES

[1] Blacklisted IPs dataset. http://www.myip.ms/browse/blacklist. Last visit in may 2015.

[2] Cisco Systems NetFlow Services Export Version 9. http://www.rfc-base.org/txt/rfc-3954.txt/. Last visit in May 2015.

[3] Google Battery Historian tool. https://github.com/google/battery-historian. Last visit in may 2015.

[4] IDC - Smartphone OS market share 2014. https://www.idc.com/. Last visit in May 2015.

[5] Statista - Number of available applications in the Google Play Store from December 2009 to February 2015. http://www.statista.com/. Last visit in May 2015.

[6] The Dansguardian tool. http://dansguardian.org/. Last visit in may 2015.

[7] The Emerging Threats open ruleset. http://www.emergingthreats.net/open-source/etopen-ruleset. Last visit in may 2015.

[8] The Mininet emulator. http://mininet.org/. Last visit in may 2015.

[9] The Netfilter firewall. http://www.netfilter.org/. Last visit in september 2014.

[10] The Squid proxy. http://www.squid-cache.org/. Last visit in may 2015.

[11] The Suricata IDS/IPS. http://suricata-ids.org/. Last visit in may 2015.

[12] A. Carroll and G. Heiser. An Analysis of Power Consumption in a Smartphone. In *Proceedings of 2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*, 2010.

[13] A. Cheung, O. Arden, S. Madden, and A. C. Myers. Automatic Partitioning of Database Applications. *PVLDB*, 5(11):1471–1482, 2012.

[14] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song. NetworkProfiler: Towards Automatic Fingerprinting of Android Apps. In *Proceedings of the IEEE INFOCOM 2013, Turin, Italy, April 14-19, 2013*, pages 809–817, 2013.

[15] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. Gaur, M. Conti, and R. Muttukrishnan. Android Security: A Survey of Issues, Malware Penetration and Defenses. *Communications Surveys Tutorials, IEEE*, PP(99):1–1, 2015.

[16] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 543–546, 2014.

[17] J. Ferrer, F. Chicano, and E. Alba. Estimating Software Testing Complexity. *Information & Software Technology*, 55(12):2125–2139, 2013.

[18] G. Gibb, H. Zeng, and N. McKeown. Outsourcing Network Functionality. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, pages 73–78. ACM, 2012.

[19] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking App Behavior Against App Descriptions. In *Proceedings of the 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 1025–1035, 2014.

[20] G. Hurel, R. Badonnel, A. Lahmadi, and O. Festor. Towards Cloud-Based Compositions of Security Functions for Mobile Devices. In *Proceedings of the 14th IFIP/IEEE International Symposium on Integrated Network Management (IM 2015)*. IEEE, 2015.

[21] C. Kilinc, T. Booth, and K. Andersson. WallDroid: Cloud Assisted Virtualized Application Specific Firewalls for the Android OS. In *Proceedings of the 11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2012)*, pages 877–883, 2012.

[22] A. Lahmadi, F. Beck, E. Finickel, and O. Festor. A Platform for the Analysis and Visualization of Network Flow Data of Android Environments. In *Demonstrations of the 14th IFIP/IEEE International Symposium on Integrated Network Management (IM 2015)*. IEEE, 2015.

[23] J. Lin, S. Amini, J. I Hong, N. Sadeh, J. Lindqvist, and J. Zhang. Expectation and Purpose: Understanding Users' Mental Models of Mobile App Privacy Through Crowdsourcing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, pages 501–510. ACM, 2012.

[24] A. Mylonas, A. Kastania, and D. Gritzalis. Delegate the Smartphone User? Security Awareness in Smartphone Platforms. *Computers & Security*, 34:47–66, 2013.

[25] J. Oberheide, K. Veeraraghavan, E. Cooke, J. Flinn, and F. Jahanian. Virtualized In-Cloud Security Services for Mobile Devices. In *Proceedings of the 1st Workshop on Virtualization in Mobile Computing (MobiVirt'08)*, page 31–35, 2008.

[26] M. La Polla, F. Martinelli, and D. Sgandurra. A Survey on Security for Mobile Devices. *IEEE Communications Surveys and Tutorials*, 15(1):446–471, 2013.

[27] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying Middlebox Policy Enforcement using SDN. In *ACM SIGCOMM 2013 Conference, SIGCOMM'13, Hong Kong, China, August 12-16, 2013*, pages 27–38, 2013.

[28] A. Sapio, Y. Liao, M. Baldi, G. Ranjan, F. Risso, A. Tongaonkar, R. Torres, and A. Nucci. Per-user Policy Enforcement on Mobile Apps Through Network Functions Virtualization. In *Proceedings of the 9th ACM Workshop on Mobility in the Evolving Internet Architecture, MobiArch 2014, Maui, HI, USA, September 11, 2014*, pages 37–42, 2014.

[29] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *ACM SIGCOMM 2012 Conference, SIGCOMM '12, Helsinki, Finland - August 13 - 17, 2012*, pages 13–24, 2012.

[30] R. Stevens, J. Ganz, V. Filkov, P. T. Devanbu, and H. Chen. Asking For (and About) Permissions Used by Android Apps. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pages 31–40, 2013.

[31] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. ProfileDroid: Multi-Layer Profiling of Android Applications. In *The 18th Annual International Conference on Mobile Computing and Networking, Mobicom'12, Istanbul, Turkey, August 22-26, 2012*, pages 137–148, 2012.